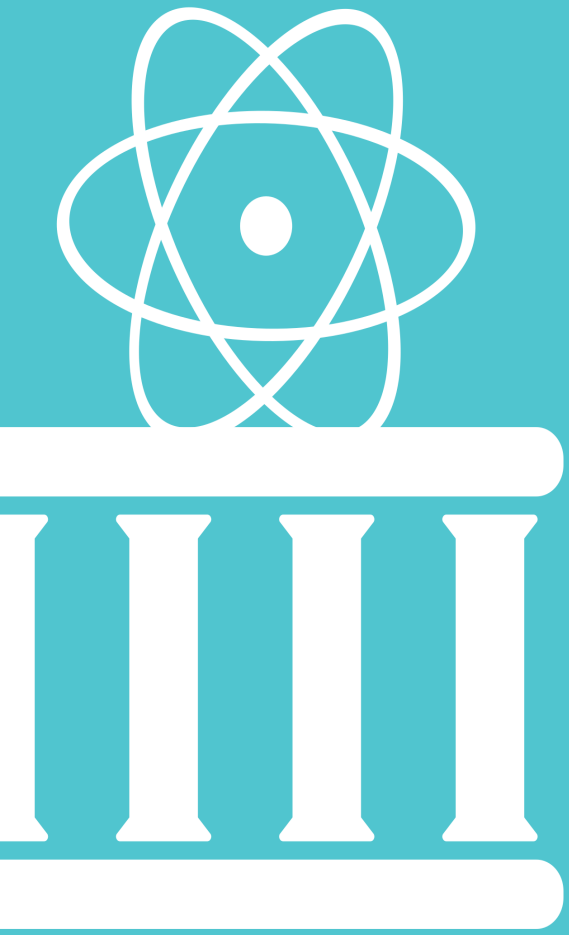
Foundations of *High-Performance* React Applications



by

Thomas Hintz

Foundations of High-Performance React Applications

Thomas Hintz

Contents

Preface

Welcome to *Foundations of High-Performance React Applications* where we build our own simplified version of React. We'll use our React to gain an understanding of the real React and how to build high-performance applications with it.

This book is based on the first chapter of the book *High-Performance React*. If you enjoy this book and you want to learn more practical ways to utilize the foundations we'll learn here and get a more detailed blueprint for creating high performance React applications, then be sure to check out *High-Performance React*.

This book is not intended to be an introduction to React or JavaScript. While it might be useful to beginners, this book assumes familiarity with both JavaScript and React.

And while this book only specifically addresses React-DOM, the foundations apply equally to React-Native and other React implementations because they are all based on the same core React library and algorithms.

The code in this book is clear and simple so as to best communicate the algorithms we'll be exploring. It is not intended to be used in production, but it is functional. I think

you'll likely find it useful to follow along by writing the code yourself. It will help you better understand how it works, and even more critically, it will allow you to play with it and test how the algorithms work with your own examples.

Even if you don't write out the code yourself and, instead, read through this book more like a novel, I believe the fundamentals will still stick with you and provide value in your React programs-to-come.

I'm very excited to take you on this journey with me and, so, now it's time to learn what lies at the very foundation of React.

Acknowledgments

First, I'd like to thank my partner Laura for always supporting me in whatever endeavors I embark upon, whether they're new, challenging, or scary. This book and my work with React wouldn't have taken place if it weren't for her support and strength.

Second, I would like to thank my friend Timothy Licata for providing invaluable feedback on earlier versions and always pushing me to explore new ways of using Emacs, such as writing this book.

And last, I would like to thank the wider JavaScript and React community for providing many years of work to build upon, specifically Rodrigo Pombo's "Build Your Own React" for being the inspiration for a lot of this book's content.

Introduction

When I first began to learn how to bake bread, a recipe told me what to do. It listed some ingredients, told me how to combine them, and prescribed times of rest. It gave me an oven temperature and a period of wait. It gave me mediocre bread of wildly varying quality. I tried different recipes, but the result was always the same.

Understanding: that's what I was missing. The bread I make is now consistently good. The recipes I use are simpler and only give ratios and general recommendations for rests and waits. So, why does the bread now turn out better?

Before it is baked, bread is a living organism. So, the way it grows, develops, and flavors depends on what you feed it, how you feed it, how you massage it, and how you care for it. If you have it grow and ferment with more yeast at a higher temperature, it overdevelops, producing too much alcohol. If you give it too much time, acidity will take over the flavor. The recipes I used initially were missing a critical ingredient: the rising temperature.

But unlike other ingredients, temperature is hard for the home cook to control. And recipes don't say exactly at which

temperature to grow the bread. My initial recipes just silently made assumptions about the temperature, which rarely worked. This means the only way to consistently make good bread is to have an understanding of how bread develops so that you can adjust the other ingredients to complement the temperature. Now the bread can tell me what to do.

While React isn't technically a living organism that can tell us what to do, it is, in its whole, a complex, abstract entity. We could learn basic recipes for how to write high-performance React code, but they wouldn't apply in all cases. And as React and things under it change, our recipes would fall out-of-date. So, like bread, to produce consistently good results we need to understand how React does what it does.

Components of React

The primary elements that make up any React program are its components. A `component` in React maintains local state and "renders" output to eventually be included in the browser's DOM. A tree of components is then created whenever a component outputs other components.

So, conceptually, React's core algorithm is very simple: it starts by walking a tree of components and building up a tree of their output. Then it compares that tree to the tree currently in the browser's DOM to find any differences between them. When it finds differences it updates the browser's DOM to match its internal tree.

But what does that actually look like? If your app is janky does that explanation point you towards what is wrong? No. It might make you wonder if maybe it is too expensive to re-render the tree or if maybe the diffing React does is slow, but you won't really know. When I was initially testing out different bread recipes I had guesses at why it wasn't working, but I didn't really figure it out until I had a deeper understanding of how making bread worked. It's time we build up our understanding of how React works so that we can start to answer our questions with solid answers.

React is centered on the `render` method. The `render` method is what walks our trees, diffs them with the browser's DOM tree, and updates the DOM as needed. But before we can look at the `render` method we have to understand its input. The input comes from `createElement`. While `createElement` itself is unlikely to be a bottleneck, it's good to understand how it works so that we can have a complete picture of the entire process. The more black-boxes we have in our mental model the harder it will be for us to diagnose performance problems.

Markup in JavaScript: JSX

`createElement`, however, takes as input something that is probably not familiar to us since we usually work in JSX, which is the last element of the chain in this puzzle and the first step in solving it. While not strictly a part of React, it is almost universally used with it. And if we understand it, `createElement` will then be less of a mystery since we'll be able to connect all the dots.

JSX is not valid HTML or JavaScript but its own language compiled by a compiler, like Babel. The output of that compilation is valid JavaScript that represents the original markup.

Before JSX or similar compilers, the typical way of injecting HTML into the DOM was via directly utilizing the browser's DOM APIs or by setting `innerHTML`. This was very cumbersome. The code's structure did not match the structure of the HTML that it output which made it hard to quickly understand what the output of a piece of code would be. So naturally programmers have been endlessly searching for better ways to mix HTML with JavaScript.

And this brings us to JSX. It is nothing new, nothing complicated. Forms of it have been made and used long before React adopted it. Now let's see if we can discover JSX for ourselves.

To start with, we need to create a data-structure – let's call it JavaScript Markup (JSM) – that both represents a DOM tree and can also be used to insert one into the browser's DOM. And to do that we need to understand what a tree of DOM nodes is constructed of. What parts do you see here?

```
<div class="header">
   <h1>Hello</h1>
   <input type="submit" disabled />
</div>
```

I see three parts: the name of the tag, the tag's properties, and its children.

Name:	'div', 'h1', 'input'
Props:	'class', 'type', 'disabled'
Children:	<h1>, <input>, Hello

Now how could we recreate that in JavaScript?

In JavaScript, we store lists of things in arrays, and key/value properties in objects. Luckily for us, JavaScript even gives us literal syntax for both so we can easily make a compact DOM tree with our own notation.

This is what I'm thinking:

JSM - JavaScript Markup

```
['div', { 'className': 'header' },
   [['h1', {}, ['Hello']],
    ['input', { 'type': 'submit', 'disabled': 'disabled' },
     []]
   ]
]
```

As you can see, we have a clear mapping from our notation, JSM, to the original HTML. Our tree is made up of three element arrays. The first item in the array is the tag, the second is an object containing the tag's properties, and the third is an array of its children which are all made up of the same three element arrays.

The truth is, if you stare at it long enough, although the mapping is clear, how much fun would it be to read and write that on a consistent basis? I can assure you, it is not fun. But it has the advantage of being easy to insert into the DOM. All you need to do is write a simple recursive function that ingests our data structure and updates the DOM accordingly. We'll get back to that.

So now we have a way to represent a tree of nodes and we (theoretically) have a way to get those nodes into the DOM. But if we are being honest with ourselves, while functional, it isn't a pretty notation nor easy to work with.

And this is where our object of study enters the scene. JSX is just a notation that a compiler takes as input and outputs in

its place a tree of nodes nearly identical to the notation we came up with! And if you look back to our notation you can see that you can easily embed in a node arbitrary JavaScript expressions wherever you want. As you may have realized, that's exactly what the JSX compiler does when it sees curly braces!

There are three main differences between JSM and the real output of the JSX compiler: it uses objects instead of arrays, it inserts calls to React.createElement on children, and spreads the children instead of containing them in an array. Here is what real JSX compiler output looks like:

```
React.createElement(
   'div',
   { className: 'header' },
   React.createElement('h1', {}, 'Hello'),
   React.createElement(
      'input',
      { type: 'submit', 'disabled': 'disabled' })
);
```

As you can see, it is very similar to our JSM data-structure and, for the purposes of this book, we'll use JSM, as it's a bit easier to work with. A JSX compiler also does some validation and escapes input to prevent cross-site scripting attacks. In practice though, it would behave the same in our areas of study and we'll keep things simple by leaving out those aspects of the JSX compiler.

So now that we've worked through JSX we're ready to tackle

`createElement`, the next item on our way to building our own React.

Getting Ready to Render with createElement

React's `render` expects to consume a tree of element objects in a specific, uniform format. `createElement` is the method by which we achieve that objective. `createElement` will take as input JSM and output a tree of objects compatible with `render`.

React expects nodes defined as JavaScript objects that look like this:

```
{
   type: NODE_TYPE,
   props: {
      propA: VALUE,
      propB: VALUE,
      ...
      children: STRING | ARRAY
   }
}
```

That is, an object with two properties: `type` and `props`. The `props` property contains all the properties of the node. The node's `children` are also considered part of its properties. The

full version of React's `createElement` includes more properties, but they are not relevant to our study here.

```
function createElement(node) {
   // if array (our representation of an element)
   if (Array.isArray(node)) {
      const [ tag, props, children ] = node;
      return {
         type: tag,
         props: {
             ...props,
             children: children.map(createElement)
         }
      };
   }

   // primitives like text or number
   return {
      type: 'TEXT',
      props: {
           nodeValue: node,
           children: []
      }
   };
}
```

Our `createElement` has two main parts: complex elements and primitive elements. The first part tests whether `node` is a complex node (specified by an array) and then generates an `element` object based on the input node. It recursively calls `createElement` to generate an array of children elements. If the node is not complex then we generate an element of type 'TEXT' which we use for all primitives, like strings and numbers. We call the output of `createElement` a tree of `elements` (surprise).

That's it. Now we have everything we need to actually begin the process of rendering our tree to the DOM!

Render: Putting Elements on the Screen

There are now only two major puzzles remaining in our quest for our own React. The next piece is `render`. How do we go from our JSM tree of nodes to actually displaying something on screen? We do this by exploring the `render` method.

The signature for our `render` method should be familiar to you:

```
function render(element, container)
```

This is the same signature as that of React itself. We begin by just focusing on the initial render. In pseudocode it looks like this:

```
function render(element, container) {
   const domElement = createDOMElement(element);
   setProps(element, domElement);
   renderChildren(element, domElement);
   container.appendChild(domElement);
```

Our DOM element is created first. Then we set the properties, render the children elements, and finally append the whole tree to the container.

Now that we have an idea of what to build we'll work on expanding the pseudocode until we have our own fully functional `render` method by using the same general algorithm that React uses. In our first pass we'll focus on the initial render and ignore reconciliation.

Reconciliation is basically React's "diffing" algorithm. We'll be exploring it after we work out the initial render.

```
function render(element, container) {
   const { type, props } = element;

   // create the DOM element
   const domElement = type === 'TEXT' ?
     document.createTextNode(props.nodeValue) :
     document.createElement(type);

   // set its properties
   Object.keys(props)
     .filter((key) => key !== 'children')
     .forEach((key) => domElement[key] = props[key]);

   // render its children
   props.children.forEach((child) =>
      render(child, domElement));

   // add our tree to the DOM!
   container.appendChild(domElement);
}
```

The `render` method starts by creating the DOM element. Then we need to set its properties. To do this we first need to filter out the `children` property and then we simply loop over the keys, setting each property directly. Following that, we render each of the children by looping over them and recursively calling `render` on each child with the `container` set to the current DOM element (which is each child's parent).

Now we can go all the way from our JSX-like notation to a rendered tree in the browser's DOM! But so far we can only add things to our tree. To be able to remove and modify the tree we need one more part: reconciliation.

Reconciliation, or How React Diffs

This is a tale of two trees, the two trees that people most often talk about when talking about React's "secret sauce": the virtual DOM and the browser's DOM tree. This idea is what originally set React apart. React's reconciliation is what allows you to program declaratively. Reconciliation is what makes it so we no longer have to manually update and modify the DOM whenever our own internal state changes. In a lot of ways, it is what makes React, React.

Conceptually, the way this works is that React generates a new element tree for every render and compares the newly generated tree to the tree generated on the previous render. Where it finds differences between the trees it knows to mutate the DOM state. This is the "tree diffing" algorithm.

Unfortunately, those researching tree diffing in Computer Science have not yet produced a generic algorithm with sufficient performance for use in something like React, as the current best algorithm still runs in $O(n^3)$.

Since an $O(n^3)$ algorithm isn't going to cut it in the real-world, the creators of React instead use a set of heuristics to deter-

mine what parts of the tree have changed. Understanding the heuristics currently in use and how the React tree diffing algorithm works in general can help immensely in detecting and fixing React performance bottlenecks. And beyond that it can help one's understanding of some of React's quirks and usage. Even though this algorithm is internal to React and can be changed anytime, its details have leaked out in some ways and, overall, are unlikely to change in major ways without larger changes to React itself.

According to the React documentation the diffing algorithm is O(n) and is based on two major components:

- Elements of differing types will yield different trees
- You can hint at tree changes with the `key` prop.

In this section we'll focus on the first part: differing types.

In this book we won't be covering keys in depth, but you'll see why it's very important to follow the guidance from React's documentation that keys are stable, predictable, and unique.

The approach we'll take here is to integrate the heuristics that React uses into our `render` method. Our implementation will be very similar to how React itself does it and we'll discuss React's actual implementation later when we talk about Fibers.

Before we get into the code changes that implement the heuristics, it is important to remember that React *only* looks at an element's type, existence, and key. It does not do any other diffing. It does not diff props. It does not diff sub-trees of modified parents.

While keeping that in mind, here is an overview of the algorithm we'll be implementing in the `render` method. `element` is the element from the current tree and `prevElement` is the corresponding element in the tree from the previous render.

```
if (!element && prevElement)
   // delete dom element
else if (element && !prevElement)
   // add new dom element, render children
else if (element.type === prevElement.type)
   // update dom element, render children
else if (element.type !== prevElement.type)
   // replace dom element, render children
```

Notice that in every case, except deletion, we still call `render` on the element's children. And while it's possible that the children will have their associated DOM elements reused, their `render` methods will still be invoked.

Now, to get started with our render method we must make some modifications to our previous render method. First, we need to be able to store and retrieve the previous render tree. Then, we need to add code to compare parts of the tree to decide if we can reuse DOM elements from the previous render tree. And last, we need to return a tree of elements

that can be used in the next render as a comparison and to reference the DOM elements that we create. These new element objects will have the same structure as our current elements but we'll add two new properties: `domElement` and `parent`. `domElement` is the DOM element associated with our synthetic element and `parent` is a reference to the parent DOM element.

Here we begin by adding a global object that will store our last render tree, keyed by the `container`. `container` refers to the browser's DOM element that will be the parent for all of the React derived DOM elements. This parent DOM element can only be used to render one tree of elements at a time, so it works well to use it as a key for `renderTrees`.

```
const renderTrees = {};
function render(element, container) {
   const tree =
      render_internal(element, container,
        renderTrees[container]);
   // render complete, store the updated tree
   renderTrees[container] = tree;
}
```

As you can see, the change we made is to move the core of our algorithm into a new function called `render_internal` and pass in the result of our last render to `render_internal`.

Now that we have stored our last render tree, we can go ahead and update our render method with the heuristics for reusing the DOM elements. We name it `render_internal` because it is

what controls the rendering, but it now takes an additional argument: the `prevElement`. `prevElement` is a reference to the corresponding `element` from the previous render and contains a reference to its associated DOM element and parent DOM element. If it's the first render or if we are rendering a new node or branch of the tree, then `prevElement` will be `undefined`. If, however, `element` is `undefined` and `prevElement` is defined, then we know we need to delete a node that previously existed.

```
function render_internal(element, container, prevElement) {
   let domElement, children;
   if (!element && prevElement) {
      removeDOMElement(prevElement);
      return;
   } else if (element && !prevElement) {
      domElement = createDOMElement(element);
   } else if (element.type === prevElement.type) {
      domElement = prevElement.domElement;
   } else { // types don't match
      removeDOMElement(prevElement);
      domElement = createDOMElement(element);
   }
   setDOMProps(element, domElement, prevElement);
   children =
      renderChildren(element, domElement, prevElement);

   if (!prevElement ||
       domElement !== prevElement.domElement) {
      container.appendChild(domElement);
   }

   return {
      domElement: domElement,
      parent: container,
      type: element.type,
```

```
      props: {
         ...element.props,
         children: children
      }
   };
}
```

The only time we shouldn't set DOM properties on our element and render its children is when we are deleting an existing DOM element. We use this observation to group the calls for `setDOMProps` and `renderChildren`. Choosing when to append a new DOM element to the container is also part of the heuristics. If we can reuse an existing DOM element, then we do this, but if the element type has changed or if there was no corresponding existing DOM element, then, and only then, do we append a new DOM element. This ensures the actual DOM tree isn't being replaced every time we render, only the elements that change are being replaced.

In the real React, when a new DOM element is appended to the DOM tree, React would invoke `componentDidMount` or schedule `useEffect`.

Next up we'll go through all the auxiliary methods that complete the implementation.

Removing a DOM element is straightforward; we just `removeChild` on the parent element. Before removing the element, React would invoke `componentWillUnmount` and schedule the cleanup function for `useEffect`.

```
function removeDOMElement(prevElement) {
   prevElement.parent.removeChild(prevElement.domElement);
}
```

In creating a new DOM element, we just need to branch if we are creating a text element since the browser API differs slightly. We also populate the text element's value, as the API requires the first argument to be specified even though later on when we set props we'll set it again. This is where React would invoke `componentWillMount` or schedule `useEffect`.

```
function createDOMElement(element) {
   return element.type === 'TEXT' ?
     document.createTextNode(element.props.nodeValue) :
     document.createElement(element.type);
}
```

To set the props on an element, we first clear all the existing props and then loop through the current props, setting them accordingly. Of course, we filter out the `children` prop since we use that elsewhere and it isn't intended to be set directly.

```
function setDOMProps(element, domElement, prevElement) {
   if (prevElement) {
      Object.keys(prevElement.props)
          .filter((key) => key !== 'children')
          .forEach((key) => {
              domElement[key] = ''; // clear prop
          });
   }
   Object.keys(element.props)
     .filter((key) => key !== 'children')
     .forEach((key) => {
        domElement[key] = element.props[key];
   });
}
```

React is more intelligent about only updating or removing props that need to be updated or removed.

This algorithm for setting props does not correctly handle events, which must be treated specially. For this exercise, that detail is not important and we leave it out for simplicity.

For rendering children we use two loops. The first loop removes any elements that are no longer being used. This would happen when the number of children is decreased. The second loop starts at the first child and then iterates through all of the children of the parent element, calling `render_internal` on each child. When `render_internal` is called, the corresponding

previous element in that position is passed to `render_internal`, or `undefined` if there is no corresponding element, like when the list of children has grown.

```
function renderChildren(element, domElement,
      prevElement = { props: { children: [] }}) {
   const elementLen = element.props.children.length;
   const prevElementLen = prevElement.props.children.length;
   // remove now unused elements
   for (let i = elementLen; i < prevElementLen - elementLen;
        i++) {
      removeDOMElement(element.props.children[i]);
   }
   // render existing and new elements
   return element.props.children.map((child, i) => {
      const prevChild = i < prevElementLen ?
         prevElement.props.children[i] : undefined;
      return render_internal(child, domElement, prevChild);
   });
}
```

It's very important to understand the algorithm used here because this is essentially what happens in React when incorrect keys are used, like using a list index for a key. And this is why keys are so critical to high performance (and correct) React code. For example, in our algorithm here, if you removed an item from the front of the list, you may cause every element in the list to be created anew in the DOM if the types no longer match up. In this book we won't be incorporating keys, but it's actually only a minor difference in determining which `child` gets paired with which `prevChild`. Otherwise this is effectively the same algorithm React uses when rendering lists of children.

Example of `renderChildren` *2nd loop when the 1st element has been removed. In this case, the trees for all of the children will be torn down and rebuilt.*

i	child Type	prevChild Type
0	span	div
1	input	span
2	-	input

There are a few things to note here. First, it is important to pay attention to when React will be removing a DOM element from the tree and adding a new one, as this is when the related lifecycle events, or hooks, are invoked. And invoking those lifecycle methods, or hooks, and the whole process of tearing down and building up a component is expensive. So again, if you use a bad key, like the algorithm here simulates, you'll be hitting a major performance bottleneck since React will not only be replacing DOM elements in the browser but also tearing down and rebuilding the trees of child components.

Fibers: Splitting up Render

The actual React implementation used to look very similar to what we've built so far, but with React 16 this has changed dramatically with the introduction of Fibers. Fibers is a name that React gives to discrete units of work during the render process. And the React reconciliation algorithm was changed to be based on small units of work instead of one large, potentially long-running call to `render`. This means that React is now able to process just part of the render phase, pause to let the browser take care of other things, and resume again. This is the underlying change that enables the experimental Concurrent Mode as well as runs most hooks without blocking the render.

But even with such a large change, the underlying algorithms that decide how and when to render components are the same. And, when not running in Concurrent Mode, the effect is still the same, as React still does the render phase in one block. So, using a simplified interpretation that doesn't include all the complexities of breaking up the process into chunks enables us to see more clearly how the process works as a whole. At

this point, bottlenecks are much more likely to occur from the underlying algorithms and not from the Fibers specific details.

Putting it all together

Now that we've explored how React renders your components, it's time to finally create some components and use them!

```
const SayNow = ({ dateTime }) => {
   return ['h1', {}, [`It is: ${dateTime}`]];
};

const App = () => {
   return ['div', { 'className': 'header' },
            [SayNow({ dateTime: new Date() }),
             ['input',
              { 'type': 'submit', 'disabled': 'disabled' },
              []]
            ]
           ];
}

render(createElement(App()),
       document.getElementById('root'));
```

We are creating two components that output JSM, as we defined it earlier. We create one component prop for the `SayNow` component: `dateTime`. It gets passed from the `App` component. The `SayNow` component prints out the `DateTime` passed in to it. You might notice that we are passing props the same way one does in the real React, and it just works!

The next step is to call render multiple times.

```
setInterval(() =>
   render(createElement(App()),
          document.getElementById('root')),
   1000);
```

If you run the code above you'll see the DateTime display being updated every second. And if you watch in your dev tools, or if you profile the run, you'll see that the only part of the DOM that gets updated or replaced is the part that changes (aside from the DOM props). We now have a working version of our own React.

This implementation is designed for teaching purposes and has some known bugs, like always updating the DOM props, along with other issues. Fundamentally, it functions the same as React, but if you want to use it in a more production-like setting, it would take a lot more development.

Conclusion

Of course our version of React elides over many details that React must contend with, like starting a re-render from where state changes and event handlers. To build high-performance React applications, however, the most important piece to understand is how and when React renders components, which is what we have learned in creating our own mini version of React.

At this point, you should now have an understanding of how React works. You should now understand why using a good `key` is so critical, what it actually means to have React render a tree of components, and how React chooses when to replace a node or re-use one. If your React application is performing poorly you can think about which part of the algorithm or heuristics might be the issue.

Now, there is a lot more to explore. Like, how do you track down the cause of a performance bottleneck? Or, how do you use the React APIs in a performant way? These types of questions should be easier to track down and understand with the foundations we covered. So I hope this is just the start of your high-performance React journey.